# UNDERSTANDING PACKAGES

**AGENDA**

**Packaging Up Your Classes**

**Standard Packages**
Standard Classes Encapsulating the Primitive Data Types
Converting between Primitive Type Values and Strings
Converting Objects to Values
Autoboxing Values of Primitive Types

**Controlling Access To Class Members**
Using Access Attributes
Specifying Access Attributes

**Nested Classes**

## Packaging Up Your Classes

Packages are implicit in the organization of the standard classes as well as your own programs. Essentially, a *package* is a uniquely named collection of classes. The primary reason for grouping classes in packages is to avoid possible name clashes with your own classes when you are using prewritten classes in an application. The names used for classes in one package do not interfere with the names of classes in another package or your program because the class names in a package are all qualified by the package name. Thus, the String class you have been using is in the java.lang package, so the full name of the class is java.lang.String. Every class in Java is contained in a package. All of the standard classes in Java are contained within a set of packages.

Putting one of your classes in a named package is very simple. You just add a package statement as the first statement in the source file containing the class definition. Note that it must always be the *first* statement. A *package statement* consists of the keyword package followed by the package name and is terminated by a semicolon. If you want the classes in a package to be accessible outside the package, you must declare the class using the keyword public in the first line of your class definition.
For example, to include the Line class in a package called Geometry, the contents of the file Line.java needs to be:

```
package Geometry;
  public class Line {
          // Details of the class definition
}
```

Each class that you want to include in the package Geometry must contain the same package statement at the beginning, and you must save all the files for the classes in the package in a directory with the same name as the package, that is, Geometry.

Note the use of the public keyword in the definition of the Line class. This makes the class accessible generally. If you omit the public keyword from the class definition, the class is accessible only from methods in classes that are in the Geometry package. Note that you also need to declare the constructors and methods in the class as public if you want them to be accessible from outside of the package.

**Packages and the Directory Structure**

You can specify a package name as any sequence of names separated by periods. For example, you might have developed several collections of classes dealing with geometry, perhaps one that works with 2D shapes and another with 3D shapes. In this case you might include the class Sphere in a package with the statement

```
package Geometry.Shapes3D;
```

and the class for circles in a package using the statement

```
package Geometry.Shapes2D;
```

In this situation, the files containing the classes in the Geometry.Shapes3D packages are expected to be in the directory Shapes3D and the files containing the classes in the Geometry.Shapes2D packages are expected to be in the directory Shapes2D. Both of these directories must be subdirectories of a directory with the name Geometry.

**Adding Classes from a Package to Your Program**

Assuming the classes have been defined with the public keyword, you can add all or any of the classes in a named package to the code in your program by using an import *statement.* For example, to make available all the classes in the package Geometry.Shapes3D to a source file, you just need to add the following import statement to the beginning of the file:

```
import Geometry.Shapes3D.*;      // Include all classes from this package
```

The keyword import is followed by the specification of what you want to import. The wildcard *, following the period after the package name, selects all the classes in the package, rather like selecting all the files in a directory. Now you can refer to any public class in the package just by using the class name.

It's usually better to import just the names from a package that your code references. If you want to add a particular class rather than an entire package, you specify its name explicitly in the import statement:

```
import Geometry.Shapes3D.Sphere;          // Include the class Sphere
```

This includes only the Sphere class in the source file. By using a separate import statement for each individual class from the package, you ensure that your source file includes only the classes that you need.

## Standard Packages

All of the standard classes that are provided with Java are stored in standard packages. There is a substantial and growing list of standard packages.

| PACKAGE | DESCRIPTION |
|---|---|
| `java.lang` | Contains classes that are fundamental to Java (e.g., the `Math` class) and all of these are available in your programs automatically. You do not need an import statement to include them. |
| `java.nio.file` | Contains classes supporting file I/O in conjunction with the classes in the `java.io` package. This package is new in JDK 7. |
| `java.awt` | Contains classes that support Java's graphical user interface (GUI). Although you can use these classes for GUI programming, it is almost always easier and better to use the alternative Swing classes. |
| `javax.swing` | Provides classes supporting the "Swing" GUI components. These are not only more flexible and easier to use than the `java.awt` equivalents, but they are also implemented largely in Java with minimal dependency on native code. |
| `java.awt.event` | Contains classes that support event handling. |
| `java.awt.geom` | Contains classes for drawing and operating with 2D geometric entities. |

| PACKAGE | DESCRIPTION |
|---|---|
| `javax.swing.border` | Classes to support generating borders around Swing components. |
| `javax.swing.event` | Classes supporting event handling for Swing components. |
| `java.applet` | Contains classes that enable you to write applets—programs that are embedded in a web page. |
| `java.util` | Contains classes that support a range of standard operations for managing collections of data, accessing date and time information, and analyzing strings. |

**Standard Classes Encapsulating the Primitive Data Types**

The following classes enable you to define objects that encapsulate values of each of the primitive data types in Java.

| Primitive Types | Wrapper Classes |
|---|---|
| boolean | java.lang.Boolean |
| byte | java.lang.Byte |
| char | java.lang.Character |
| double | java.lang.Double |
| float | java.lang.Float |
| int | java.lang.Integer |
| long | java.lang.Long |
| short | java.lang.Short |

These are all contained in the package java.lang along with quite a few other classes, such as the String and StringBuffer classes. Each of these classes encapsulates a value of the corresponding primitive type and includes methods for manipulating and interrogating objects of the class, as well as a number of very useful static methods that provide utility functions for the underlying primitive types.

**Converting between Primitive Type Values and Strings**

Each class provides a static toString() method to convert a value of the corresponding primitive type to a String object.

```
        java.lang.Integer
public static String toString(int i)
```

There is also a non-static toString() method in each class that returns a String representation of a class object.

```
        java.lang.Integer
public String toString()
```

There are methods to convert from a String object to a primitive type. For example, the static parseInt() member in the Integer class accepts a String representation of an integer as an argument and returns the equivalent value as type int.

```
        java.lang.Integer
public static int parseInt(String s)
           throws NumberFormatException
```

All the standard classes encapsulating numerical primitive types and the Boolean class define static methods to parse strings and return a value of the corresponding primitive type. You have the methods parseShort(), parseByte(), parseInt(), and parseLong() in the classes for integer types, parseFloat() and parseDouble() for fl oating-point classes, and parseBoolean() for Boolean.

```
        java.lang.Float
public static float parseFloat(String s)
             throws NumberFormatException
```

The classes for primitive numerical types and the Boolean class defines a static method valueOf() that converts a string to an object of the class type containing the value represented by the string.

```
        java.lang.Float
public static Float valueOf(String s)
            throws NumberFormatException
```

**Converting Objects to Values**

Each class encapsulating a primitive data value also defines a xxxValue() method (where xxx is the corresponding primitive type name) that returns the value that is encapsulated by an object as a value of the corresponding primitive type. For example, if you have created an object number of type Double that encapsulates the value 1.14159, then the expression number.doubleValue()results in the value 1.14159 as type double.

```
        java.lang.Integer

public byte byteValue()
public double doubleValue()
public float floatValue()
public int intValue()
public long longValue()
public short shortValue()
```

**Autoboxing Values of Primitive Types**

Circumstances can arise surprisingly often where you want to pass values of a primitive type to a method that requires the argument to be a reference to an object. The compiler supplies automatic conversions of primitive values to the corresponding class type when circumstances permit this. This can arise when you pass a value of type int to a method where the parameter type is type Integer, for example. Conversions from a primitive type to the corresponding class type are called *boxing conversions*, and automatic conversions of this kind are described as *autoboxing*.

The compiler also inserts unboxing conversions when necessary to convert a reference to an object of a wrapper class for a primitive type such as double to the value that it encapsulates. The compiler does this by inserting a call to the xxxValue() method for the object.

```
public class AutoboxingInAction          {
   public static void main(String[] args)     {

        int[] values = { 3, 97, 55, 22, 12345 };
// Array to store Integer objects
        Integer[] objs = new Integer[values.length];
// Call method to cause boxing conversions
        for(int i = 0 ; i < values.length ; ++i) {

                objs[i] = boxInteger(values[i]);
                }
// Use method to cause unboxing conversions
        for(Integer intObject : objs) {
                unboxInteger(intObject);
        }
 }
// Method to cause boxing conversion
        public static Integer boxInteger(Integer obj) {
                return obj;
        }
// Method to cause unboxing conversion
        public static void unboxInteger(int n) {
                System.out.println(“value = “ + n);
        }
```

You have defined the boxInteger() method with a parameter type of type Integer. When you call this method in the first for loop in main(), you pass values of type int to it from the values array. Because the boxInteger() method requires the argument to be a reference to an object of type Integer, the compiler arranges for autoboxing to occur by inserting a boxing conversion to convert the integer value to an object of type Integer. The method returns a reference to the object that results, and you store this in the Integer[] array objs.

The second for loop in main() passes each reference to an Integer object from the objs array to the unboxInteger() method. Because you have specified the method parameter type as type int, the method cannot accept a reference to an object of type Integer as the argument directly. The compiler inserts an unboxing conversion to obtain the value of type int that the object encapsulates. This value is then passed to the method, and you output it.

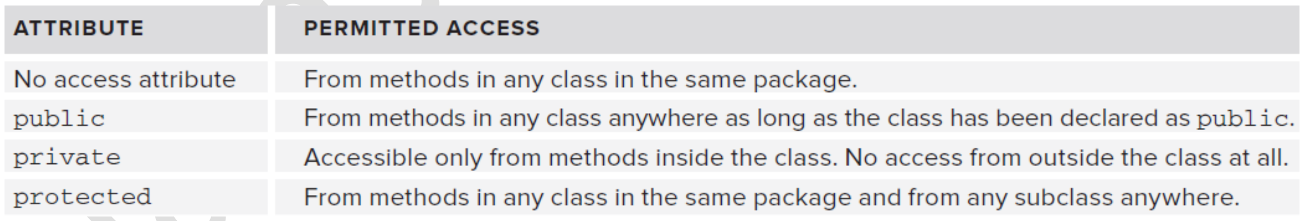
Autoboxing is particularly useful when you need to insert values of primitive types into a collection — later you learn about the collection classes that are available in the class libraries.

## Controlling Access To Class Members

How can you control the accessibility of class members from outside the class — from a method in another class in other words. Variables and methods within one class are accessible from other classes is a bit more complicated. It depends on what *access attributes* you have specified for the members of a class, whether the classes are in the same package, and whether you have declared the class as public.

### Using Access Attributes

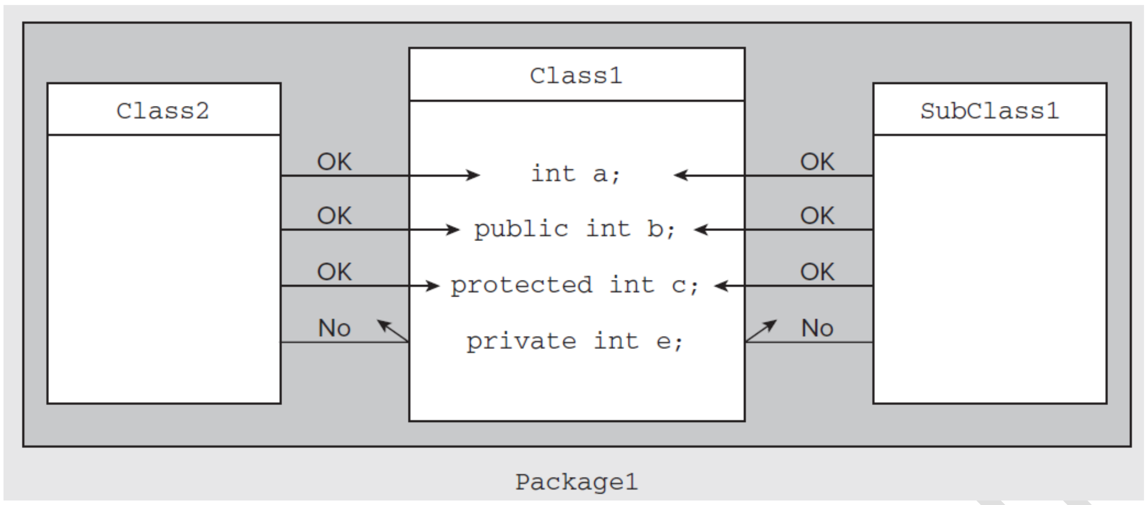
Let's start by considering classes that are in the same package. Within a given package, any class has direct access to any other class name in the same package. The name of a class in one package can be accessed from a class in another package only if the class to be accessed is declared as public. Classes not declared as public can be accessed only by classes within the same package. You have four possibilities when specifying an access attribute for a class member, and each possibility has a different effect overall. The options you have for specifying the accessibility of a variable or a method in a class are found in Table



| ATTRIBUTE | PERMITTED ACCESS |
|---|---|
| No access attribute | From methods in any class in the same package. |
| `public` | From methods in any class anywhere as long as the class has been declared as `public`. |
| `private` | Accessible only from methods inside the class. No access from outside the class at all. |
| `protected` | From methods in any class in the same package and from any subclass anywhere. |

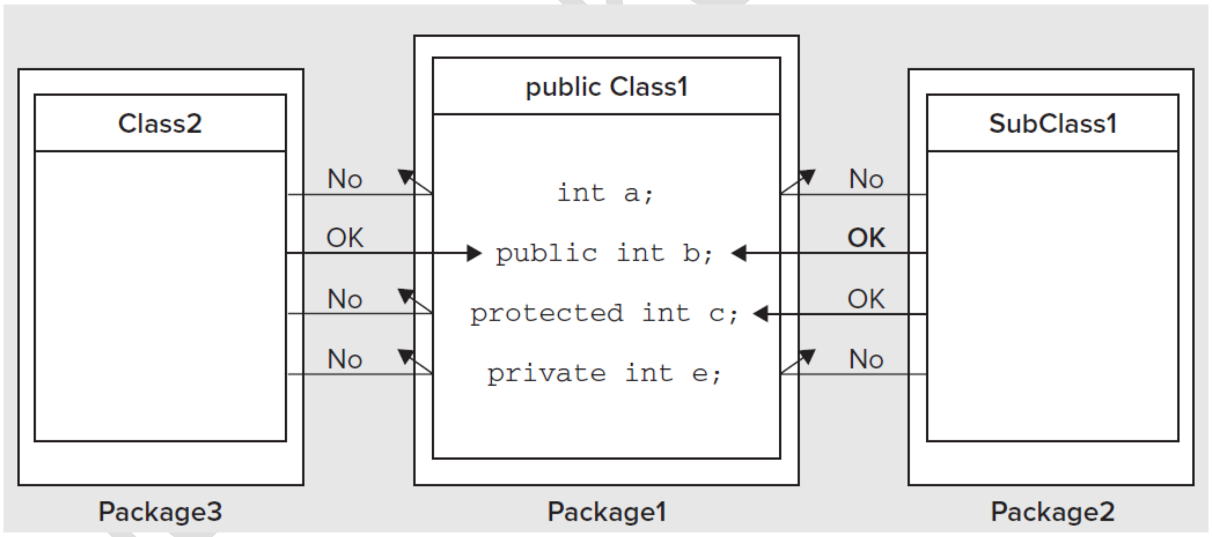
The table shows you how the access attributes you set for a class member determine the parts of the Java environment from which you can access it. Note that public, private, and protected are all keywords. Specifying a member as public makes it completely accessible, and at the other extreme, making it private restricts access to members of the same class.

The following figure shows the access allowed between classes within the same package.



Within a package such as package1 in the above figure, only the private members of the class Class1 can't be directly accessed by methods in other classes in the same package. If you declare a class member to be private, it can be accessed only by methods in the same class.

A class definition must have an access attribute of public if it is to be accessible from outside the package that contains it. The following figure shows the situation where the classes seeking access to the members of a public class are in different packages.

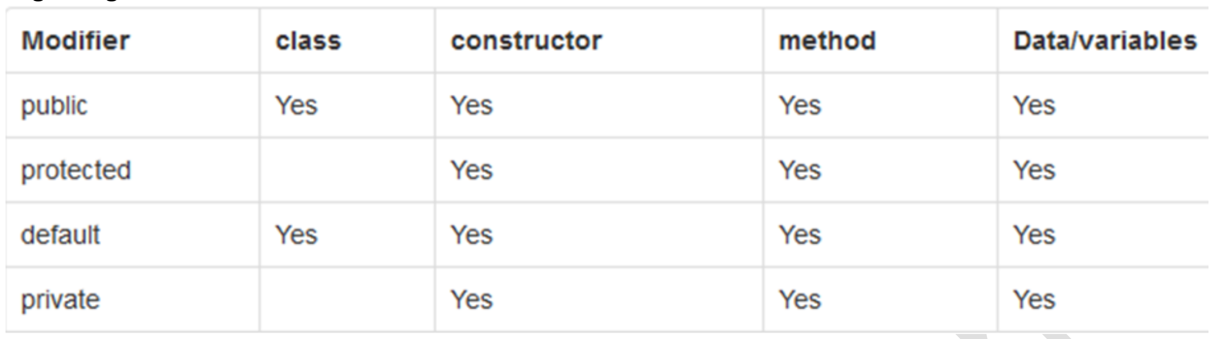


Here access is more restricted. The only members of Class1 that can be accessed from an ordinary class, Class2, in another package, are those specified as public. Keep in mind that the class Class1 must also have been defined with the attribute public for this to be the case. A class that is not defined as public cannot be accessed at all from a class in another package.

From a subclass of Class1 that is in another package, the members of Class1 without an access attribute cannot be reached, and neither can the private members — these can never be accessed externally under any circumstances.

**Specifying Access Attributes**

To specify an access attribute for a class member, you just add the appropriate keyword to the beginning of the declaration.



| Modifier | class | constructor | method | Data/variables |
|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes |
| protected | | Yes | Yes | Yes |
| default | Yes | Yes | Yes | Yes |
| private | | Yes | Yes | Yes |

Here is the Point class with access attributes defined for its members:

```
import static java.lang.Math.sqrt;
public class Point {
// Create a point from its coordinates
        public Point(double xVal, double yVal) {
                x = xVal;
                y = yVal;
        }
// Create a Point from an existing Point object
        public Point(fi nal Point aPoint) {
        x = aPoint.x;
        y = aPoint.y;
  }
// Move a point
        public void move(double xDelta, double yDelta) {
// Parameter values are increments to the current coordinates
        x += xDelta;
        y += yDelta;
       }
// Calculate the distance to another point
        public double distance(fi nal Point aPoint) {
                return sqrt((x - aPoint.x)*(x - aPoint.x)+(y - aPoint.y)*(y - aPoint.y));
        }
// Convert a point to a string
        public String toString() {
                return Double.toString(x) + ", " + y; // As "x, y"
        }
// Coordinates of the point
        private double x;
        private double y;
}
```

Now the instance variables x and y cannot be accessed or modified from outside the class, as they are private. The only way these can be set or modified is through methods within the class, either with constructors or the move() method. If it is necessary to obtain the values of x and y from outside the class, as it might well be in this case, a simple function does the trick. For example

```
public double getX() {
        return x;
}
```

This makes x freely available, but prevents modification of its value from outside the class. In general, such methods are referred to as *accessor* methods and usually have the form getXXX(). Methods that allow a private data member to be changed are called *mutator* methods and are typically of the form setXXX(), where a new value is passed as an argument. For example:

```
public void setX(double inputX) {
        x = inputX;
}
```

Note: It may seem odd to use a method to alter the value of a private data member when you could just make it public. The main advantage of using a method in this way is that you can apply validity checks on the new value that is to be set and prevent inappropriate values from being assigned. Of course, if you really don't want to allow the value of a private member to be changed, you don't include a mutator method for the class.

## Nested Classes

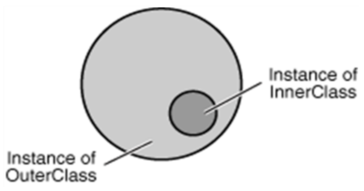
You can put the definition of one class inside the definition of another class. The inside class is called a *nested class*. When you define a nested class, it is a member of the enclosing class in much the same way as the other class members. A nested class can have an access attribute just like other class members, and the accessibility from outside the enclosing class is determined by the attributes in the same way. The enclosing class here is referred to as a *top-level class*.

```
public class Outside {
// Nested class
        public class Inside {
                // Details of Inside class...
        }
// More members of Outside class...
}
```

### Using a Non-Static Nested Class

Non-static nested classes are also called inner classes. Inner classes are associated with an instance of the enclosing class. Thus, you must first create an instance of the enclosing class to create an instance of an inner class. Non-static nested classes have access to the fields of the enclosing class, even if they are declared private. An instance of Inner Class can exist only within an instance of Outer Class.

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }}
```



It is very important to state that the instance of the Inner class doesn't exist in isolation, rather it is bounded by and associated with an instance of their enclosing top-level-class, and this impose a slight restriction on the inner nested classes which is that they can't have any static members that normal top-level and static nested classes can.

The nested class here has meaning only in the context of an object of type OuterClass. This is because the 'InnerClass' is not declared as a static member of the class OuterClass. Until an object of type OuterClass has been created, you can't create any InnerClass objects.

For example, suppose you create an object with the following statement:

OuterClass outer = new OuterClass();

No objects of the nested class, Inside, are created. If you now want to create an object of the type of the nested class, you must refer to the nested class type using the name of the enclosing class as a qualifier. For instance, having declared an object of type OuterClass, you can create an object of type InnerClass as follows:

OuterClass.InnerClass inner = outer.new InnerClass(); // Define a nested class object

You are creating an object of type InnerClass in the context of the object OuterClass.

Within non static methods that are members of OuterClass, you can use the class name InnerClass without any qualification, as it is automatically qualified by the compiler with the this variable. So you could create a new InnerClass object from within a method of the object OuterClass:

InnerClass  inner = new InnerClass(); // Define a nested class object

This statement is equivalent to:
this.InnerClass inner = this.new InnerClass(); // Define a nested class object

Note: You typically use nested classes to define objects that at least have a strong association with objects of the enclosing class type, and often there is a tight coupling between the two. A further use for nested classes is for grouping a set of related classes under the umbrella of an enclosing class.

**Static Nested Classes**

To make objects of a nested class type independent of objects of the enclosing class type, you can declare the nested class as static:

When you declare an inner class `static`, you do not require an object of the outer class to create the inner class and you are restricted from accessing a non-`static` outer class from an object of a nested class.

static nested classes can’t directly access the members of the class that they are defined in (simply because they are static members, and in static class members can only access other static members directly), but they can do so using an object instance of the enclosing top-level-class.

To obtain an instance of a static nested class from within code defined outside the enclosing class, one needs to follow the normal instance creation mechanism in java as the static class exist in isolation from their enclosing class and they are not associated with any instance of their enclosing class.

**Local Classes**

Java also allows you to create a local inner classes that exist within method scopes. A local inner class is a class created within the scope of a method as opposed to just being within a class. You use a local inner class to return a reference when you're implementing an interface and to help solve some very complex programming problems whereby a class might be the only means of sufficiently expressing the level of complexity required.

Local classes are like inner (non-static nested) classes that are defined inside a method or scope block ({ ... }) inside a method.

Local classes can only be accessed from inside the method. Local classes can access members (fields and methods) of its enclosing class just like regular inner classes.

Local classes can also access local variables inside the same method or scope block, provided these variables are declared final.

Local classes can also be declared inside static methods. In that case the local class only has access to the static parts of the enclosing class. Local classes cannot contain all kinds of static declarations (constants are allowed - variables declared static final), because they are non-static in nature - even if declared inside a static method.

**Anonymous Classes**

Java also allows you to create an anonymous inner classes that exist within method scopes. You use an anonymous inner class to return a reference when you're implementing an interface and to help solve some very complex programming problems.

The anonymous inner class provides exactly the same functionality as an inner class except that it is shorthand and quicker to write because you write the class as an expression within the method or scope.

Anonymous classes are classed without a class name. They are typically declared as either subclasses of an existing class, or as implementations of some interface.

Anonymous classes are defined when they are instantiated. Anonymous classes can have direct access to all of the class members of the top-level-class in which they are defined.

You can define these classes not just within a method, but even within an argument to a method.

```
class Ferrari {
public void drive() {
```

```
System.out.println("Ferrari");
}
}
class Car {
Ferrari p = new Ferrari() {
public void drive() {
System.out.println("anonymous Ferrari");
}
};
}
```

Let’s look at what’s in the preceding code:
- We define two classes, Ferrari and Car.
- Ferrari has one method, drive().
- Car has one instance variable, declared as type Ferrari. That’s it for Car. Car has no methods.

The Ferrari reference variable refers not to an instance of Ferrari, but to an instance of an anonymous (unnamed) subclass of Ferrari.

In car class, line 2 starts out as an instance variable declaration of type Ferrari. But instead of looking like this:

```
Ferrari p = new Ferrari(); // notice the semicolon at the end
```

there’s a curly brace at the end of line 2, where a semicolon would normally be.

```
Ferrari p = new Ferrari()  { // a curly brace, not a semicolon
```

You can read line 2 as saying,
Declare a reference variable, p, of type Ferrari. Then declare a new class that has no name, but that is a subclass of Ferrari. And here’s the curly brace that opens the class definition...

Line 3, then, is actually the first statement within the new class definition. And what is it doing? Overriding the drive() method of the superclass Ferrari. This is the whole point of making an anonymous inner class—to override one or more methods of the superclass!

Here’s where you have to pay attention: line 6 includes a curly brace closing off the anonymous class definition (it’s the companion brace to the one on line 2), but there’s more! Line 6 also has the semicolon that ends the statement started on line 2—the statement where it all began—the statement declaring and initializing the Ferrari reference variable. And what you’re left with is a Ferrari reference to a brand-new instance of a brand-new, anonymous (no name) subclass of Ferrari.